MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo 521 June 1979

A TRUTH MAINTENANCE SYSTEM

Jon Doyle

Abstract: To choose their actions, reasoning programs must be able to make assumptions and subsequently revise their beliefs when discoveries contradict these assumptions. The *Truth Maintenance System* (TMS) is a problem solver subsystem for performing these functions by recording and maintaining the reasons for program beliefs. Such recorded reasons are useful in constructing explanations of program actions and in guiding the course of action of a problem solver. This paper describes (1) the representations and structure of the TMS, (2) the mechanisms used to revise the current set of beliefs, (3) how dependency-directed backtracking changes the current set of assumptions, (4) techniques for summarizing explanations of beliefs, (5) how to organize problem solvers into "dialectically arguing" modules, (6) how to revise models of the belief systems of others, and (7) methods for embedding control structures in patterns of assumptions. We stress the need of problem solvers to choose between alternative systems of beliefs, and outline a mechanism by which a problem solver can employ rules guiding choices of what to believe, what to want, and what to do.

This research was conducted at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract number N00014-75-C-0643, and in part by NSF grant MCS77-04828.

Acknowledgements: This paper is based on a thesis submitted in partial fulfillment of the degree of Master of Science to the department of Electrical Engineering and Computer Science of the Massachusetts Institute of Technology on May 12, 1977. Sections 6 and 7 contain material not reported in that thesis. Although I have held the views expressed in Section 1.1 for several years, I found my prior attempts at explaining them unsatisfactory, so these views appear here for the first time as well. I thank Gerald Jay Sussman (thesis advisor), Johan de Kleer, Scott Fahlman, Philip London, David McAllester, Drew McDermott, Marvin Minsky, Howard Shrobe, Richard M. Stallman, Guy L. Steele, Jr., and Alan Thompson for ideas and comments. de Kleer, Steele, Marilyn Matz, Richard Fikes, Randall Davis, Shrobe, and the referees of the journal *Artificial Intelligence* gave me valuable editorial advice. I thank the Fannie and John Hertz Foundation for supporting my research with a graduate fellowship.

Contents:

In memory of John Sheridan Mac Nerney

1. Introduction

Computer reasoning programs usually construct computational models of situations. To keep these models consistent with new information and changes in the situations being modelled, the reasoning programs frequently need to remove or change portions of their models. These changes sometimes lead to further changes, for the reasoner often constructs some parts of the model by making inferences from other parts of the model. This paper studies both the problem of how to make changes in computational models, and the underlying problem of how the models should be constructed in order to make making changes convenient. Our approach is to record the reasons for believing or using each program belief, inference rule, or procedure. To allow new information to displace previous conclusions, we employ "non-monotonic" reasons for beliefs, in which one belief depends on a lack of belief in some other statement. We use a program called the *Truth Maintenance System*[1] (TMS) to determine the current set of beliefs from the current set of reasons, and to update the current set of beliefs in accord with new reasons in a (usually) incremental fashion. To perform these revisions, the TMS traces the reasons for beliefs to find the consequences of changes in the set of assumptions.

1.1 The Essence of the Theory

Many treatments of formal and informal reasoning in mathematical logic and artificial intelligence have been shaped in large part by a seldom acknowledged view: the view that the process of reasoning is the process of deriving new knowledge from old, the process of discovering new truths contained in known truths. This view, as it is simply understood, has several severe difficulties as a theory of reasoning. In this section, I propose another, quite different view about the nature of reasoning. I incorporate some new concepts into this view, and the combination overcomes the problems exhibited by the conventional view.

Briefly put, the problems with the conventional view of reasoning stem from the *monotonicity* of the sequence of states of the reasoner's beliefs: his beliefs are true, and truths never change, so the only action of reasoning is to augment the current set of beliefs with more beliefs. This monotonicity leads to three closely related problems involving commonsense reasoning, the frame problem, and control. To some extent, my criticisms here of the conventional view of reasoning will be amplifications of Minsky's [36] criticisms of

1. As we shall see, this term not only sounds like Orwellian Newspeak, but also is probably a misnomer. The name stems from historical accident, and rather than change it here, I retain it to avoid confusion in the literature.

the logistic approach to problem solving.

One readily recalls examples of the ease with which we resolve apparent contradictions involving our commonsense beliefs about the world. For example, we routinely make assumptions about the permanence of objects and the typical features or properties of objects, yet we smoothly accommodate corrections to the assumptions and can quickly explain our errors away. In such cases, we discard old conclusions in favor of new evidence. Thus, the set of our commonsense beliefs changes non-monotonically.

Our beliefs of what is current also change non-monotonically. If we divide the trajectory of the temporally evolving set of beliefs into discrete temporal situations, then at each instant the most recent situation is the set of current beliefs, and the preceding situations are past sets of beliefs. Adjacent sets of beliefs in this trajectory are usually closely related, as most of our actions have only a relatively small set of effects. The important point is that the trajectory does not form a sequence of monotonically increasing sets of beliefs, since many actions change what we expect is true in the world. Since we base our actions on what we currently believe, we must continually update our current set of beliefs. The problem of describing and perfoming this updating efficiently is sometimes called the *frame problem*. In connection with the frame problem, the conventional view suffers not only from monotonicity, but also from *atomicity*, as it encourages viewing each belief as an isolated statement, related to other beliefs only through its semantics. Since the semantics of beliefs are usually not explicitly represented in the system, if they occur there at all, atomicity means that these incremental changes in the set of current beliefs are difficult to compute.

The third problem with the conventional view actually subsumes the problem of commonsense reasoning and the frame problem. The problem of control is the problem of deciding what to do next. Rather than make this choice blindly, many have suggested that we might apply the reasoner to this task as well, to make inferences about which inferences to make. This approach to the problem of control has not been explored much, in part because such control inferences are useless in monotonic systems. In these systems, adding more inference rules or axioms just increases the number of inferences possible, rather than preventing some inferences from being made. One gets the unwanted inferences together with new conclusions confirming their undesireability.

Rather than give it up, we pursue this otherwise attractive approach, and make the deliberation required to choose actions a form of reasoning as well. For our purposes, we take the desires and intentions of the reasoner to be represented in his set of current beliefs as beliefs about his own desires and intention. We also take the set of inference rules by which the reasoning process occurs to be represented as beliefs about the reasoner's own computational structure. By using this self-referential, reflexive representation of the reasoner, the inference rules become rules for self-modification of the reasoner's set of beliefs (and hence his desires and intentions as well). The control problem of choosing which inference rule to follow takes the form "Look at yourself as an object (as a set of beliefs), and choose what (new set of beliefs) you would like to become."

The language of such inference rules, and the language for evaluating which self-change to make, are for the most part outside the language of inference rules

encouraged by the conventional view of reasoning. For example, when the current set of beliefs is inconsistent, one uses rules like "Reject the smallest set of beliefs possible to restore consistency" and "Reject those beliefs which represent the simplest explanation of the inconsistency." These sorts of rules are all we have, since we cannot infallibly analyze errors or predict the future, yet these rules are non-monotonic, since they lead to removing beliefs from the set of current beliefs.

To repeat, one source of each of these problems is the monotonicity inherent in the conventional view of reasoning. I now propose a different view, and some new concepts which have far reaching consequences for these issues.

Rational thought is the process of finding reasons for attitudes.

To say that some attitude (such as belief, desire, intent, or action) is rational is to say that there is some acceptable reason for holding that attitude. Rational thought is the process of finding such acceptable reasons. Whatever purposes the reasoner may have, such as solving problems, finding answers, or taking action, it operates by constructing reasons for believing things, desiring things, intending things, or doing or willing things. The actual attitude in the reasoner occurs only as a by-product of constructing reasons. The current set of beliefs and desires arises from the current set of reasons for beliefs and desires, reasons phrased in terms of other beliefs and desires. When action is taken, it is because some reason for the action can be found in terms of the beliefs and desires of the actor. I stress again, the only *real* component of thought is the current set of reasons - the attitudes such as beliefs and desires arise from the set of reasons, and have no independent existence.

One consequence of this view is that to study rational thought, we should study justified belief or reasoned argument, and ignore questions of truth. Truth enters into the study of extra-psychological rationality and into what commonsense truisms we decide to supply to our programs, but truth does not enter into the narrowly psychological rationality by which our programs operate.

Of course, this sort of basic rationality is simpler to realize than human belief. Humans exhibit "burn-in" phenomena in which long-standing beliefs come to be believed independently of their reasons, and humans sometimes undertake "leaps of faith" which vault them into self-justifying sets of beliefs, but we will not study these issues here. Instead, we restrict ourselves to the more modest goal of making rational programs in this simpler sense.

The view stated above entails that for each statement or proposition P just one of two states obtains: Either

(A) P has at least one currently acceptable (*valid*) reason, and is thus a member of the current set of beliefs, or

(B) P has no currently acceptable reasons (either no reasons at all, or only unacceptable ones), and is thus not a member of the current set of beliefs.

If P falls in state (A), we say that P is *in* (the current set of beliefs), and otherwise, that P is *out* (of the current set of beliefs). These states are not symmetric, for while reasons can be constructed to make P *in*, no reason can make P *out*. (At most, it can make ¬P *in* as well.)

This shows that the proposed view also succumbs to monotonicity problems, for the set of reasons grows monotonically, which (with the normal sense of "reason") leads to only monotonic increases in the set of current beliefs. To solve the problem of monotonicity, we introduce novel meanings for the terms "a reason" and "an assumption."

Traditionally, a reason for a belief consists of a set of other beliefs, such that if each of these basis beliefs is held, so also is the reasoned belief. To get off the ground, this analysis of reasons requires either circular arguments between beliefs (and the appropriate initial state of belief) or some fundamental type of belief which grounds all other arguments. The traditional view takes these fundamental beliefs, often called assumptions (or premises), as believed without reason. On this view, the reasoner makes changes in the the current set of beliefs by removing some of the current assumptions and adding some new ones.

To conform with the proposed view, we introduce meanings for "reason" and "assumption" such that assumptions also have reasons. A *reason* (or justification) for a belief consists of an ordered pair of sets of other beliefs, such that the reasoned belief is *in* by virtue of this reason only if each belief in the first set is *in*, and each belief in the second set is *out*. An *assumption* is a current belief one of whose valid reasons depends on a non-current belief, that is, has a non-empty second set of antecedent beliefs. With these notions we can create "ungrounded" yet reasoned beliefs by making assumptions. (E.g. give P the reason ({},{¬P}).) We can also effect non-monotonic changes in the set of current beliefs by giving reasons for some of the *out* statements used in the reasons for current assumptions. (E.g. to get rid of P, justify ¬P.) We somewhat loosely say that when we justify some *out* belief supporting an assumption, (e.g. ¬P), we are *denying* or *retracting* the assumption (P).

These new notions solve the monotonicity problem. Following from this solution we find ways of treating the commonsense reasoning, frame, and control problems plaguing the conventional view of reasoning. Commonsense default expectations we represent as new-style assumptions. Part of the frame problem, namely how to non-monotonically change the set of current beliefs, follows from this non-monotonic notion of reason. However, much of the frame problem (e.g. how to give the "laws of motion" and how to retrieve them efficiently) lies outside the scope of this discussion. The control problem can be dealt with partially by embedding the sequence of procedural states of the reasoner in patterns of assumptions. We will treat this idea, and the rest of the control problem, in more detail later.

Other advantages over the conventional view also follow. One of these advantages involves how the reasoner retracts assumptions. With the traditional notion of assumption, retracting assumptions was unreasoned. If the reasoner removed an assumption from the current set of beliefs, the assumption remained out until the reasoner specifically put it back into the set of current beliefs, even if changing circumstances obviated the value of removing this belief. The new notions introduce instead the *reasoned retraction of assumptions*. This means that the reasoner retracts an assumption only by giving a reason

for why it should be retracted. If later this reason becomes invalid, then the retraction is no longer effective and the assumption is restored to the current set of beliefs.

The reasoned retraction of assumptions helps in formulating a class of backtracking procedures which revise the set of current assumptions when inconsistencies are discovered. The paradigm procedure of this sort we call dependency-directed backtracking after Stallman and Sussman [53]. It is the least specialized procedure for revising the current set of assumptions in the sense that it only operates on the reasons for beliefs, not on the form or content of the beliefs. In short, it traces backwards through the reasons for the conflicting beliefs, finds the set of assumptions reached in this way, and then retracts one of the assumptions with a reason involving the other assumptions. (We describe the procedure in detail later.) Dependency-directed backtracking serves as a template for more specialized revision procedures. These specialized procedures are necessary in almost all practical applications, and go beyond the general procedure by taking the form of the beliefs they examine into account when choosing which assumption to reject.

1.2 Basic Terminology

The TMS records and maintains arguments for potential program beliefs, so as to distinguish, at all times, the current set of program beliefs. It manipulates two data structures: *nodes*, which represent beliefs, and *justifications*, which represent reasons for beliefs. We write St(N) to denote the statement of the potential belief represented by the node N. We say the TMS believes in (the potential belief represented by) a node if it has an argument for the node and believes in the nodes involved in the argument. This may seem circular, but some nodes will have arguments which involve no other believed nodes, and so form the base step for the definition.

As its fundamental actions, (1) the TMS can create a new node, to which the problem solving program using the TMS can attach the statement of a belief (or inference rule, or procedure, or data structure). The TMS leaves all manipulation of the statements of nodes (for inference, representation, etc.) to the program using the TMS. (2) It can add (or retract) a new justification for a node, to represent a step of an argument for the belief represented by the node. This argument step usually represents the application of some rule or procedure in the problem solving program. Usually, the rules or procedures also have TMS nodes, which they include in the justifications they create. (3) Finally, the TMS can mark a node as a *contradiction*, to represent the inconsistency of any set of beliefs which enter into an argument for the node.

A new justification for a node may lead the TMS to believe in the node. If the TMS did not believe in the node previously, this may in turn allow other nodes to be believed by previously existing but incomplete arguments. In this case, the TMS invokes the *truth maintenance* procedure to make any necessary revisions in the set of beliefs. The TMS revises the current set of beliefs by using the recorded justifications to compute non-circular arguments for nodes from premises and other special nodes, as described later. These non-circular arguments distinguish one justification as the *well-founded supporting*

justification of each node representing a current belief. The TMS locates the set of nodes to update by finding those nodes whose well-founded arguments depend on changed nodes.

The program using the TMS can indicate the inconsistency of the beliefs represented by certain currently believed nodes by using these nodes in an argument for a new node, and by then marking the new node as a contradiction. When this happens, another process of the TMS, *dependency-directed backtracking*, analyzes the well-founded argument of the contradiction node to locate the *assumptions* (special types of nodes defined later) occurring in the argument. It then makes a record of the inconsistency of this set of assumptions, and uses this record to change one of the assumptions. After this change, the contradiction node is no longer believed. We explain this process in Section 4.

The TMS employs a special type of justification, called a *non-monotonic justification*, to make tenative guesses. A non-monotonic justification bases an argument for a node not only on current belief in other nodes, as occurs in the most familiar forms of deduction and reasoning, but also on lack of current belief in other nodes. For example, one might justify a node N-1 representing a statement P on the basis of lack of belief in node N-2 representing the statement ¬P. In this case, the TMS would hold N-1 as a current belief as long as N-2 was not among the current beliefs, and we would say that it had assumed belief in N-1. More generally, by an *assumption* we mean any node whose well-founded support is a non-monotonic justification.

As a small example of the use of the TMS, suppose that a hypothetical office scheduling program considers holding a meeting on Wednesday. To do this, the program assumes that the meeting is on Wednesday. The inference system of the program includes a rule which draws the conclusion that due to regular commitments, any meeting on Wednesday must occur at 1:00 P.M. However, the fragment of the schedule for the week constructed so far has some activity scheduled for that time already, and so another rule concludes the meeting cannot be on Wednesday. We write these nodes and rule-constructed justifications as follows:

Node	*Statement*	*Justification*	*Comment*
N-1	DAY(M) = WEDNESDAY	(SL () (N-2))	*an assumption*
N-2	DAY(M) ≠ WEDNESDAY		*no justification yet*
N-3	TIME(M) = 13:00	(SL (R-37 N-1) ())	

The above notation for the justifications indicates that they belong to the class of *support-list* (SL) justifications. Each of these justifications consists of two lists of nodes. A SL-justification is a *valid* reason for belief if and only if each of the nodes in the first list is believed and each of the nodes in the second list is not believed. In the example, if the two justifications listed above are the only existing justifications, then N-2 is not a current belief since it has no justifications at all. N-1 is believed since the justification for N-1 specifies that this node depends on the lack of belief in N-2. The justification for N-3 shows that N-3 depends on a (presumably believed) node R-37. In this case, R-37 represents a rule acting on (the statement represented by) N-1.

Subsequently another rule (represented by a node R-9) acts on beliefs about the

day and time of some other engagement (represented by the nodes N-7 and N-8) to reject the assumption N-1.

N-2 DAY(M) ≠ WEDNESDAY (SL (R-9 N-7 N-8) ())

To accomodate this new justification, the TMS will revise the current set of beliefs so that N-2 is believed, and N-1 and N-3 are not believed. It does this by tracing "upwards" from the node to be changed, N-2, to see that N-1 and N-3 ultimately depend on N-2. It then carefully examines the justifications of each of these nodes to see that N-2's justification is valid (so that N-2 is *in*). From this it follows that N-1's justification is invalid (so N-1 is *out*), and hence that N-3's justification is invalid (so N-3 is *out*).

2. Representation of Reasons for Beliefs

2.1 States of Belief

A node may have several justifications, each justification representing a different reason for believing the node. These several justifications comprise the node's *justification-set*. The node is believed if and only if at least one of its justifications is *valid*. We described the conditions for validity of SL-justifications above, and shortly will introduce and explain the other type of justification used in the TMS. We say that a node which has at least one valid justification is *in* (the current set of beliefs), and that a node with no valid justifications is *out* (of the current set of beliefs). We will alternatively say that each node has a *support-status* of either *in* or *out*. The distinction between *in* and *out* is not that between *true* and *false*. The former classification refers to current possession of valid reasons for belief. *True* and *false*, on the other hand, classify statements according to truth value independent of any reasons for belief.

In the TMS, each potential belief to be used as a hypothesis or conclusion of an argument must be given its own distinct node. When uncertainty about some statement (e.g. P) exists, one must (eventually) provide nodes for both the statement and its negation. Either of these nodes can have or lack well-founded arguments, leading to a four-element belief set (similar to the belief set urged by Belnap [2]) of neither P nor ¬P believed, exactly one believed, or both believed.

The literature contains many proposals for using three-element belief sets of *true*, *false*, and *unknown*. With no notion of justified belief, these proposals have some attraction. I urge, however, that systems based on a notion of justified belief should forego three-valued logics in favor of the four-valued system presented here, or risk a confusion of truth with justified belief. Users of justification-based three-valued systems can avoid problems if they take care to interpret their systems in terms of justifications rather than truth-values, but the danger of confusion seems greater when the belief set hides this distinction. One might argue that holding contradictory beliefs is just a transient situation, and that any stable situation uses only three belief states: *true* - only P believed, *false* - only ¬P believed, and *unknown* - neither believed. But the need for the four-element system

cannot be dismissed so easily. Since we make the process of revising beliefs our main interest, we concern ourselves with those processes which operate during the transient situation. For hard problems and tough decisions, these "transient" states can be quite long-lived.

2.2 Justifications

Justifications, as recorded in the TMS, have two parts: the external form of the justification with significance to the problem solver, and the internal form of significance to the TMS. For example, a justification might have the external form (Modus-Ponens A A⊃B) and have the internal form (SL (N-1 N-2 N-3) ()), supposing that N-1 represents the rule Modus-Ponens, N-2 represents A, and N-3 represents A⊃B. The TMS never uses or examines the external forms of justifications, but merely records them for use by the problem solver in constructing externally meaningful explanations. Henceforth, we will ignore these external forms of justifications.

Although natural arguments may use a wealth of types of argument steps or justifications, the TMS forces one to fit all these into a common mold. The TMS employs only two (internal) forms for justifications, called *support-list* (SL) and *conditional-proof* (CP) justifications. These are inspired by the typical forms of arguments in natural deduction inference systems, which either add or subtract dependencies from the support of a proof line. A proof in such a system might run as follows:

Line	*Statement*	*Justification*	*Dependencies*
1.	A⊃B	Premise	{1}
2.	B⊃C	Premise	{2}
3.	A	Hypothesis	{3}
4.	B	MP 1,3	{1,3}
5.	C	MP 2,4	{1,2,3}
6.	A⊃C	Discharge 3,5	{1,2}

Each step of the proof has a line number, a statement, a justification, and a set of line numbers on which the statement depends. Premises and hypotheses depend on themselves, and other lines depend on the set of premises and hypotheses derived from their justifications. The above proof proves A⊃C from the premises A⊃B and B⊃C by hypothesizing A and concluding C via two applications of Modus Ponens. The proof of A⊃C ends by discharging the assumption A, which frees the conclusion of dependence on the hypothesis but leaves its dependence on the premises.

This example displays justifications which sum the dependencies of some of the referenced lines (as in line 4) and subtract the dependencies of some lines from those of other lines (as in line 6). The two types of justifications used in the TMS account for these effects on dependencies. A support-list justification says that the justified node depends on each node in a set of other nodes, and in effect sums the dependencies of the referenced nodes. A conditional-proof justification says that the node it justifies depends on the

validity of a certain hypothetical argument. As in the example above, it subtracts the dependencies of some nodes (the hypotheses of the hypothetical argument) from the dependencies of others (the conclusion of the hypothetical argument). Thus we might rewrite the example in terms of TMS justifications as follows (here ignoring the difference between premises and hypotheses, and ignoring the inference rule MP):

N-1	A⊃B	(SL () ())	*Premise*
N-2	B⊃C	(SL () ())	*Premise*
N-3	A	(SL () ())	*Premise*
N-4	B	(SL (N-1 N-3) ())	*MP*
N-5	C	(SL (N-2 N-4) ())	*MP*
N-6	A⊃C	(CP N-5 (N-3) ())	*Discharge*

CP-justifications, which will be explained in greater detail below, differ from ordinary hypothetical arguments in that they use two lists of nodes as hypotheses, the *in*hypotheses and the *out*hypotheses. In the above justification for N-6, the list of *in*hypotheses contains just N-3, and the list of *out*hypotheses is empty. This difference results from our use of non-monotonic justifications, in which arguments for nodes can be based both on *in* and *out* nodes.

2.3 Support-list Justifications

To repeat the definition scattered throughout the previous discussion, the support-list justification has the form

(SL <*in*list> <*out*list>),

and is valid if and only if each node in its *in*list is *in*, and each node in its *out*list is *out*. The SL-justification form can represent several types of deductions. With empty *in*list and empty *out*list, we say the justification forms a *premise* justification. A premise justification is always valid, and so the node it justifies will always be *in*. SL-justifications with nonempty *in*lists and empty *out*lists represent normal deductions. Each such justification represents a monotonic argument for the node it justifies from the nodes of its *in*list. We define *assumptions* to be nodes whose supporting-justification has a nonempty *out*list. These assumption justifications can be interpreted by viewing the nodes of the *in*list as comprising the reasons for wanting to assume the justified node; the nodes of the *out*list represent the specific criteria authorizing this assumption. For example, the reason for wanting to assume "The weather will be nice" might be "Be optimistic about the weather"; and the assumption might be authorized by having no reason to believe "The weather will be bad." We occasionally interpret the nodes of the *out*list as "denials" of the justified node, beliefs which imply the negation of the belief represented by the justified node.

2.4 Terminology of Dependency Relationships

I must pause to present some terminology before explaining CP-justifications. The definitions of dependency relationships introduced in this section are numerous, and the reader should consult Figures 1, 2, and 3 for examples of the definitions.

As mentioned previously, the TMS singles out one justification, called the *supporting-justification*, in the justification-set of each *in* node to form part of the non-circular argument for the node. For reasons explained shortly, all nodes have only SL-justifications as their supporting-justifications, never CP-justifications. The set of *supporting-nodes* of a node is the set of nodes which the TMS used to determine the support-status of the node. For *in* nodes, the supporting-nodes are just the nodes listed in the *in*list and *out*list of its supporting-justification, and in this case we also call the supporting-nodes the *antecedents* of the node. For the supporting-nodes of *out* nodes, the TMS picks one node from each justification in the justification-set. From SL-justifications, it picks either an *out* node from the *in*list or an *in* node from the *out*list. From CP-justifications, it picks either an *out* node from the *in*hypotheses or consequent or an *in* from the *out*hypotheses. We define the supporting-nodes of *out* nodes in this way so that the support-status of the node in question cannot change without either a change in the support-status of one of the supporting-nodes, or without the addition of a new valid justification. We say that an *out* node has no antecedents. The TMS keeps the supporting-nodes of each node as part of the node data-structure, and computes the antecedents of the node from this list.

The set of *foundations* of a node is the transitive closure of the antecedents of the node, that is, the antecedents of the node, their antecedents, and so on. This set is the set of nodes involved in the well-founded argument for belief in the node. The set of *ancestors* of a node, analogously, is the transitive closure of the supporting-nodes of the node, that is, the supporting-nodes of the node, their supporting-nodes, and so on. This set is the set of nodes which might possibly affect the support-status of the node. The ancestors of a node may include the node itself, for the closure of the supporting-nodes relation need not be well-founded. The TMS computes these dependency relationships from the supporting-nodes and antecedents of nodes.

In the other direction, the set of *consequences* of a node is the set of all nodes which mention the node in one of the justifications in their justification-set. The *affected-consequences* of a node are just those consequences of the node which contain the node in their set of supporting-nodes. The *believed-consequences* of a node are just those *in* consequences of the node which contain the node in their set of antecedents. The TMS keeps the consequences of each node as part of the node data-structure, and computes the affected- and believed-consequences from the consequences.

The set of *repercussions* of a node is the transitive closure of the affected-consequences of the node, that is, the affected-consequences of the node, their affected-consequences, and so on. The set of *believed-repercussions* of a node is the transitive closure of the believed-consequences of the node, that is, the believed-consequences of the node, their believed-consequences, and so on. The TMS computes all these

relationships from the consequences of the node.

In all of the following, I visualize the lines of support for nodes as directed upwards, so that I look up to see repercussions, and down to see foundations. I say that one node is of lower level than another if its believed-repercussions include the other node.

2.5 Conditional-proof Justifications

With this terminology, we can now begin to explain conditional-proof justifications. The exact meaning of these justifications in the TMS is complex and difficult to describe, so the reader may find this section hard going, and may benefit by referring back to it while reading Sections 3.3, 4, and 5. CP-justifications take the form

(CP <consequent> <*in*hypotheses> <*out*hypotheses>).

A CP-justification is valid if the consequent node is *in* whenever (a) each node of the *in*hypotheses is *in* and (b) each node of the *out*hypotheses is *out*. Except in a few esoteric uses described later, the set of *out*hypotheses is empty, so normally a node justified with a CP-justification represents the implication whose antecedents are the *in*hypotheses and whose consequent is the consequent of the CP-justification. Standard conditional-proofs in natural deduction systems typically specify a single set of hypotheses, which corresponds to the *in*hypotheses of a CP-justification. In the present case, the set of hypotheses must be divided into two disjoint subsets, since nodes may be derived both from some nodes being *in* and other nodes being *out*. Some deduction systems also employ multiple-consequent conditional-proofs. We forego these for reasons of implementation efficiency.

The TMS handles CP-justifications in special ways. It can easily determine the validity of a CP-justification only when the justification's consequent and *in*hypotheses are *in* and the *out*hypotheses are *out*, since determining the justification's validity with other support-statuses for these nodes may require switching the support-statuses of the hypothesis nodes and their repercussions to set up the hypothetical situation in which the validity of the conditional-proof can be evaluated. This may may require truth maintenance processing, which in turn may require validity checking of further CP-justifications, and so the whole process becomes extremely complex. Instead of attempting such a detailed analysis (for which I know no algorithms), the TMS uses the opportunistic and approximate strategy of computing SL-justifications currently equivalent to CP-justifications. At the time of their creation, these new SL-justifications are equivalent to the CP-justifications in terms of the dependencies they specify, and are easily checked for validity. Whenever the TMS finds a CP-justification valid, it computes an equivalent SL-justification by analyzing the well-founded argument for the consequent node of the CP-justification to find those nodes which are not themselves supported by any of the *in*hypotheses or *out*hypotheses but which directly enter into the argument for the consequent node along with the hypotheses. Precisely, the TMS finds all nodes N in the foundations of the consequent such that N is not one of the hypotheses or one of their repercussions, and N is either an antecedent of the consequent or an antecedent of some other node in the repercussions of the hypotheses. The *in* nodes in this set form the *in*list of the equivalent SL-justification, and the *out* nodes of the set form the *out*list of the equivalent SL-justification. The TMS attaches the list of

SL-justifications computed in this way to their parent CP-justifications, and always prefers to use these SL-justifications in its processing. The TMS checks the derived SL-justifications first in determining the support-status of a node, and uses them in explanations. It uses only SL-justifications (derived or otherwise) as supporting-justifications of nodes.

2.6 Other Types of Justifications

My experience with the TMS indicates that yet more forms of justifications would be useful. A *general-form* (GF) justification merges the above two forms into one in which the nodes in an *in*list and an *out*list are added to the result of a conditional-proof. We might notate this as

(GF <*in*list> <*out*list> <consequent> <*in*hypotheses> <*out*hypotheses>).

I also suggest a *summarization* (SUM) justification form,

(SUM <consequent> <*in*hypotheses> <*out*hypotheses>),

which abbreviates

(GF <*in*hypotheses><*out*hypotheses><consequent><*in*hypotheses><*out*hypotheses>).

This form adds the hypotheses of a conditional-proof back into the result of the conditional-proof, thus summarizing the argument for the consequent by excising the intermediate part of the argument. Section 5 explains this technique in detail. I use SUM-justifications there for expository convenience, although I have not implemented them in the TMS.

3. Truth Maintenance Mechanisms

3.1 Circular Arguments

Suppose a program manipulates three nodes as follows:

F	(= (+ X Y) 4)	...	*omitted*
G	(= X 1)	(SL (*J*) ())	
H	(= Y 3)	(SL (*K*) ()).	

(We sometimes leave statements and justifications of nodes unspecified when they are not directly relevant to the presentation. We assume that all such omitted justifications are valid.) If *J* is *in* and *K* is *out*, then the TMS will make *F* and *G* *in*, and *H* *out*. If the program then justifies *H* with

(SL (*F G*) ()),

the TMS will bring *H in*. Suppose now that the TMS makes *J out* and *K in*, leading to *G* becoming *out* and *H* remaining *in*. The program might then justify *G* with

(SL (*F H*) ()).

If the TMS now takes *K out*, the original justification supporting belief in *H* becomes invalid, leading the TMS to reassess the grounds for belief in *H*. If it makes its decision to believe a node on the basis of a simple evaluation of each of the justifications of the node, then it will leave both *G* and *H in*, since the two most recently added justifications form circular arguments for *G* and *H* in terms of each other.

These circular arguments supporting belief in nodes motivate the use of well-founded supporting justifications, since nodes imprudently believed on tenuous circular bases can lead to ill-considered actions, wasted data base searches, and illusory inconsistencies which might never have occurred without the misleading, circularly supported beliefs. In view of this problem, the algorithms of the TMS must ensure that it believes no node for circular reasons.

Purported arguments for nodes can contain essentially three different kinds of circularities. The first and most common type of circularity involves only nodes which can be taken to be *out* consistently with their justifications. Such circularities arise routinely through equivalent or conditionally equivalent beliefs and mutually constraining beliefs. The above algebra example falls into this class of circularity.

The second type of circularity includes at least one node which must be *in*. Consider, for example

F	TO-BE	(SL () (*G*))
G	¬TO-BE	(SL () (*F*)).

In the absence of other justifications, these justifications force the TMS either to make *F in* and *G out*, or *G in* and *F out*. This type of circularity can arise in certain types of sets of alternatives.

In unsatisfiable circularities, the third type, no assignment of *in* or *out* to nodes is consistent with their justifications. Consider

F	. . .	(SL () (*F*)).

With no other justifications for *F*, the TMS must make *F in* if and only if it makes *F out*, an impossible task. Unsatisfiable circularities sometimes indicate real inconsistencies in the beliefs of the program using the truth maintenance system, and can be manifest, for example, when prolonged backtracking rules out all possibilities. The current version of the TMS does not handle unsatisfiable circularities (it goes into a loop), as I removed the occasionally costly check for the presence of such circularities to increase the normal-case efficiency of the program. A robust implementation would reinstate this check. Step 5 in Section 3.2 discusses this problem in more detail.

3.2 The Truth Maintenance Process

The truth maintenance process makes any necessary revisions in the current set of beliefs when the user adds to or subtracts from the justification-set of a node. Retracting justifications presents no important problems beyond those of adding justifications, so we ignore retractions to simplify the discussion. We first outline the procedure, and then present it in greater detail. The details will not be crucial in the following, so the casual reader should read the overview and then skip to Section 4.

In outline, the truth maintenance process starts when a new justification is added to a node. Only minor bookkeeping is required if the new justification is invalid, or if it is valid but the node is already *in*. If the justification is valid and the node is *out*, then the node and its repercussions must be updated. The TMS makes a list containing the node and its repercussions, and marks each of these nodes to indicate that they have not been given well-founded support. The TMS then examines the justifications of these nodes to see if any are valid purely on the basis of unmarked nodes, that is, purely on the basis of nodes which do have well-founded support. If it finds any, these nodes are brought *in* (or *out* if all their justifications are invalid purely on the basis of well-founded nodes). Then the marked consequences of the nodes are examined to see if they too can now be given well-founded support. Sometimes, after all of the marked nodes have been examined in this way, well-founded support-statuses will have been found for all nodes. Sometimes, however, some nodes will remain marked due to circularities. The TMS then initiates a constraint-relaxation process which assigns support-statuses to the remaining nodes. Finally, after all this, the TMS checks for contradictions and CP-justifications, performs dependency-directed backtracking and CP-justification processing if necessary, and then signals the user program of the changes in support-statuses of the nodes involved in truth maintenance.

In detail, the steps of the algorithm are as follows. We enclose comments in bracket-asterisk pairs. (E.g. [* This is a comment. *])

Step 1. *Adding a new justification*: Add the new justification to the node's justification-set and add the node to the set of consequences of each of the nodes mentioned in the justification. If the justification is a CP-justification, add the node to the *CP-consequent-list* of the consequent of the CP-justification, for use in Step 6. If the node is *in*, we are done. If the node is *out*, check the justification for validity. If invalid, add to the supporting-nodes either an *out* node from the *in*list, or an *in* node from the *out*list. If valid, proceed to Step 2.

Step 2. *Updating beliefs required*: Check the affected-consequences of the node. If there are none, change the support-status to *in*, and make the supporting-nodes the sum of the *in*list and *out*list; then stop. Otherwise, make a list L containing the node and its repercussions, record the support-status of each of these nodes, and proceed to Step 3. [* We must collect all the repercussions of the node to avoid constructing circular arguments which use repercussions of a node in its supposedly well-founded supporting argument. *]

Step 3. *Marking the nodes*: Mark each node in L with a support-status of *nil*, and proceed to Step 4. [* Nodes can have a support-status of *nil* only during truth maintenance. This mark distinguishes those nodes with well-founded support from those for which well-founded support has not been determined. *]

Step 4. *Evaluating the nodes' justifications*: For each node in L, execute the following subprocedure. When all are done, proceed to Step 5.

Step 4a. *Evaluating the justification-set*: If the node is either *in* or *out*, do nothing. Otherwise, keep picking justifications from the justification-set, first the SL-justifications and then the CP-justifications, checking them for well-founded validity or invalidity (to be defined shortly) until either a valid one is found or the justification-set is exhausted. [* The TMS tries justifications in chronological order, oldest first. *] If a valid justification is found, then (1) install it as the supporting-justification (first converting it to SL form if it is a CP-justification), (2) install the supporting-nodes as in Step 2, (3) mark the node *in*, and (4) recursively perform Step 4a for all consequences of the node which have a support-status of *nil*. If only well-founded invalid justifications are found, mark the node *out*, install its supporting-nodes as in Step 1, and recursively perform Step 4a for all *nil*-marked consequences of the node. Otherwise, the processing of the node is temporarily deferred; the subprocedure is finished. [* An SL-justification is *well-founded valid* if each node in the *in*list is *in* and each node of the *out*list is *out*; it is *well-founded invalid* if some node of the *in*list is *out* or some node of the *out*list is *in*. CP-justifications are well-founded valid if all *in*hypotheses are *in*, all *out*hypotheses are *out*, and the consequent is *in*; they are well-founded invalid if all *in*hypotheses are *in*, all *out*hypotheses are *out*, and the consequent is *out*. *]

[* This step may find well-founded supporting-justifications for some nodes in L, but may leave the support-status of some nodes undetermined due to circularities in potential arguments. These leftover nodes are handled by Step 5 below. If it were not for CP-justifications, Step 4 could be dispensed with entirely, as Step 5 effectively subsumes it. However, we include Step 4 both to handle CP-justifications and to improve (we hope) the efficiency of the algorithm by getting the solidly supported nodes out of the way first. *]

Step 5. *Relaxing circularities*: For each node in L, execute the following subprocedure. On completion, proceed to Step 6.

Step 5a. *Evaluating the justification-set*: If the node is either *in* or *out*, do nothing. Otherwise, continue to select justifications from the SL-justifications (ignoring the CP-justifications) and to check them for not-well-founded validity or invalidity [* which assumes that all nodes currently marked *nil* will eventually be marked *out*, as explained shortly *] until either a valid justification is found or the justification-set is exhausted. If all justifications are invalid, mark the node *out*, install its supporting-nodes as in Step 1, and recursively perform Step 5a for all *nil*-marked consequences of the node. If a valid justification is found, then a special check must be made to see if the node already has affected-consequences. If the node does have affected-consequences, then all of them, and the node as well, must be

re-marked with *nil* and re-examined by the loop of Step 5. [* We do this because the procedure may have previously determined the support-status of some other node on the assumption that this node was *out*. *] If there are no affected-consequences, then install the valid justification as the supporting-justification, install the supporting-nodes as in Step 2, mark the node *in*, and then recursively perform Step 5a for all consequences of the node which have a support-status of *nil*. [* All justifications will be either not-well-founded valid or invalid; there is no third case. An SL-justification is not-well-founded valid if each node in the *in*list is *in* and no node of the *out*list is *in*; otherwise, it is not-well-founded invalid. This evaluation of nodes assumes that a support-status of *nil* is the same as *out*, i.e. that all currently unassigned nodes will eventually be marked *out*. *]

[* If Step 5 terminates, it finds well-founded support-statuses for all nodes in L. It will not terminate if unsatisfiable circularities exist. These circularities can be detected by checking to see if a node is its own ancestor after finding a not-well-founded valid justification for it in Step 5a. If the node is its own ancestor, an unsatisfiable circularity exists in which the argument for belief in the node depends on lack of belief in the node. Unfortunately, this sort of non-termination can also occur with a satisfiable set of nodes and justifications which requires making changes to nodes not in the list L to find this satisfiable assignment of support-statuses. For example, if *F* is *in* but not in L, and if *G* has only the justification (SL (*F*) (*G*)) and is in L, then the procedure will not be able to assign a well-founded support-status to *G* without going outside L to change the support-status of *F*. In consequence, if truth maintenance is done properly it must be non-incremental in such cases.

This relaxation procedure finds one assignment of support-statuses to nodes, but there may be several such assignments possible. A more sophisticated system would incorporate some way in which to choose between these alternatives, since guidance in what the program believes will typically produce guidance in what the program does. Some versions of the TMS (e.g. [14]) incorporated rudimentary analysis facilities, but the current version lacks any such ability. It appears that this relaxation step must be fairly blind in choosing what revision to make. Methods for choosing between alternate revisions must have some idea of what all the alternate revisions are, and these are very hard to determine accurately. One can approximate the set of alternate revisions by revisions including some particular belief, but after several such approximations this adds up to just trying some partial revision and seeing if it works out. *]

Step 6. *Checking for CP-justifications and contradictions*: Call each of the following subprocedures for each of the nodes in L, and then proceed to Step 7. [* This step attempts to derive new SL-justifications from CP-justifications, and to resolve any inconsistencies appearing in the new set of beliefs. Since Step 5 leaves all nodes in L either *in* or *out*, it may now be possible to evaluate some CP-justifications which were previously unevaluable, and to resolve newly apparent contradictions. *]

Step 6a. *Check for CP-justifications*: Do nothing if the node is *out* or has an empty

CP-consequent list. Otherwise, for each node in the CP-consequent-list, check its CP-justifications for validity, and if any are valid, derive their currently equivalent SL-justifications and justify the node with the resulting justification. If this justification is new and causes truth maintenance (Steps 1 through 5), start Step 6 over, otherwise return.

Step 6b. *Check for contradictions*: Ignore the node unless it is *in* and is marked as a contradiction, in which case call the dependency-directed backtracking system on the node. If truth maintenance (Steps 1 through 5) occurs during backtracking, start Step 6 over, otherwise return.

[* This step halts only when no new SL-justifications can be computed from CP-justifications, and no contradictions exist or can be resolved. *]

Step 7. *Signalling changes*: Compare the current support-status of each node in L with the initial status recorded in Step 2, and call the user supplied *signal-recalling functions* and *signal-forgetting functions* to signal changes from *out* to *in* and from *in* to *out*, respectively. [* The user must supply two global functions which, if not overridden by a local function that the user might attach to the changed node, are called with the changed node as the argument. However, if the user has attached local function to the changed node, the TMS will call that function instead. *]

End of the truth maintenance procedure.

For more detail, I recommend the chapter on data dependencies in [4], which presents a simplified LISP implementation of a TMS-like program along with a proof of its correctness. McAllester [30] presents an alternative implementation of a truth maintenance system with a cleaner organization than the above. Doyle [15] presents a program listing of one version of the TMS in an appendix.

3.3 Analyzing Conditional-Proofs

The *Find Independent Support* (FIS) procedure computes SL-justifications from valid CP-justifications by finding those nodes supporting the consequent which do not depend on the hypotheses. Repeating our earlier explanation of what this means, FIS finds all nodes N in the foundations of the consequent of the conditional-proof justification such that (1) N is not one of the hypotheses or one of their repercussions, and (2) N is either an antecedent of the consequent or an antecedent of some node in the repercussions of the hypotheses. The *in* nodes in this set form the *in*list of the equivalent SL-justification, and the *out* nodes of the set form the *out*list of the equivalent SL-justification.

Let (CP C IH OH) be a valid CP-justification, where C is the consequent node, IH is the list of *in*hypotheses, and OH is the list of *out*hypotheses. The steps of FIS are as follows:

Step 1. *Mark the hypotheses*: Mark each of the nodes in IH and OH with both an E (*examined*) mark and a S (*subordinates*) mark, then proceed to Step 2. [* The E mark means that the node has been examined by the procedure. We use it to make the

search through the foundations of C efficient. The S mark means that the node is either one of the hypotheses or a repercussion of one of the hypotheses. ⁎]

Step 2. *Mark the foundations*: Call the following subprocedure on C, and proceed to Step 3.

Step 2a. *Mark the repercussions of the hypotheses* If the node has an E mark, return. If the node has no E mark, mark it with the E mark, and call Step 2a on each of the antecedents of the node. If any of the antecedent nodes is marked with an S mark, mark the current node with an S mark. Finally, return. [⁎ This step marks those nodes in both the foundations of C and the repercussions of the hypotheses. ⁎]

Step 3. *Unmark the foundations*: Using a recursive scan, similar to Step 2a, remove the E marks from the foundations of C and proceed to Step 4.

Step 4. *Remark the hypotheses*: As in Step 1, mark the hypotheses with E marks, this time ignoring their S marks. Proceed to Step 5.

Step 5. *Collect the net support*: Call the following subprocedure on C and proceed to Step 6.

Step 5a. *Skip repercussions and collect support*: If the node has an E mark, return. Otherwise, mark the node with an E mark. If the node has no S mark, add it to IS if it is *in*, and to OS if it is *out*, then return. If the node has an S mark, execute step 5a on each of its antecedents, then return. [⁎ This step collects just those nodes in the foundations of C which are not hypotheses or their repercussions, and which directly support (are antecedents of) repercussions of the hypotheses. These nodes are exactly the nodes necessary to make the argument go through for C from the hypotheses. ⁎]

Step 6. *Clean up and return the result*: Repeat Step 3, removing both E and S marks, remove all marks from the nodes in IH and OH, and return the justification (SL IS OS).

End of the Find Independent Support procedure.

4. Dependency-Directed Backtracking

When the TMS makes a contradiction node *in*, it invokes dependency-directed backtracking to find and remove at least one of the current assumptions in order to make the contradiction node *out*. The steps of this process follow. As above, we enclose commentary in bracket-asterisk pairs ([⁎, ⁎]).

Step 1. *Find the maximal assumptions*: Trace through the foundations of the contradiction node *C* to find the set $S = \{A_1, ..., A_n\}$, which contains an assumption A if and only if A is in *C*'s foundations and there is no other assumption B in the foundations of *C* such that A is in the foundations of B. [⁎ We call *S* the set of the *maximal* assumptions underlying *C*. ⁎]

[⁎ Just as the TMS relies on the problem solving program to point out

inconsistencies by marking certain nodes as contradictions, it also relies on the problem solver to use non-monotonic assumptions for any beliefs to which backtracking might apply. Because the TMS does not inspect the statements represented by its nodes, it foregoes the ability, for example, to retract premise justifications of nodes. ⁂]

Step 2. *Summarize the cause of the inconsistency*: If no previous backtracking attempt on C discovered S to be the set of maximal assumptions, create a new node NG, called a *nogood*, to represent the inconsistency of S. [⁂ We call S the *nogood-set*. ⁂] If S was encountered earlier as a nogood-set of a contradiction, use the previously created nogood node. [⁂ Since C represents a false statement, NG represents

$$\mathrm{St}(A_1) \wedge \ldots \wedge \mathrm{St}(A_n) \supset \mathit{false},$$

or

(1) $$\neg\,(\mathrm{St}(A_1) \wedge \ldots \wedge \mathrm{St}(A_n))$$

by a simple rewriting. ⁂]

Justify NG with

(2) (CP C S ()).

[⁂ With this justification, NG will remain *in* even after Step 3 makes one of the assumptions *out*, since the CP-justification means that NG does not depend on any of the assumptions. ⁂]

Step 3. *Select and reject a culprit*: Select some A_i, the *culprit*, from S. Let $D_1, \ldots, D_k$ be the *out* nodes in the *out*list of A_i's supporting-justification. Select D_j from this set and justify it with

(3) (SL (NG $A_1 \ldots A_{i-1}$ $A_{i+1} \ldots A_n$) ($D_1 \ldots D_{j-1}$ $D_{j+1} \ldots D_k$)).

[⁂ If one takes these underlying D nodes as "denials" of the selected assumption, this step recalls *reductio ad absurdum*. The backtracker attempts to force the culprit *out* by invalidating its supporting-justification with the new justification, which is valid whenever the nogood and the other assumptions are *in* and the other denials of the culprit are *out*. If the backtracker erred in choosing the culprit or denial, presumably a future contradiction will involve D_j and the remaining assumptions in its foundations. However, if the *out*list of the justification (3) is nonempty, D_j will be an assumption, of higher level than the remaining assumptions, and so will be the first to be denied.

The current implementation picks the culprit and denial randomly from the alternatives, and so relies on blind search. Blind search is inadequate for all but the simplest sorts of problems, for typically one needs to make a guided choice among the alternative revisions of beliefs. I will return to this problem in Section 8. ⁂]

Step 4. *Repeat if necessary*: If the TMS finds other arguments so that the contradiction node C remains *in* after the addition of the new justification for D_j, repeat this backtracking procedure. [⁂ Presumably the previous culprit A_i will no longer be an assumption. ⁂] Finally, if the contradiction becomes *out*, then halt; or if no assumptions can be found in C's foundations, notify the problem solving program of an unanalyzable contradiction, then halt.

End of the dependency-directed backtracking procedure.

As an example, consider a program scheduling a meeting, to be held preferably at 10 A.M. in either room 813 or 801:

N-1	TIME(M) = 1000	(SL () (N-2))
N-2	TIME(M) ≠ 1000	
N-3	ROOM(M) = 813	(SL () (N-4)
N-4	ROOM(M) = 801	

With only these justifications, the TMS makes N-1 and N-3 *in* and the other two nodes *out*. Now suppose a previously scheduled meeting rules out this combination of time and room for the meeting by supporting a new node with N-1 and N-3 and then declaring this new node to be a contradiction.

N-5	CONTRADICTION	(SL (N-1 N-3) ())

The dependency-directed backtracking system traces the foundations of N-5 to find two assumptions, N-1 and N-3, both maximal.

N-6	NOGOOD N-1 N-3	(CP N-5 (N-1 N-3) ())	*here* ≡ (SL () ())
N-4	ROOM(M) = 801	(SL (N-6 N-1) ())	

The backtracker creates N-6 which means, in accordance with form (1) of Step 2,

$$\neg(\text{TIME}(M) = 1000 \wedge \text{ROOM}(M) = 813)$$

and justifies N-6 according to form (2) above. It arbitrarily selects N-3 as the culprit, and justifies N-3's only *out* antecedent, N-4, according to form (3) above. Following this, the TMS makes N-1, N-4, and N-6 *in*, and N-2, N-3, and N-5 *out*. N-6 has a CP-justification equivalent to a premise SL-justification, since N-5 depends directly on the two assumptions N-1 and N-3 without any additional intervening nodes.

A further rule now determines that room 801 cannot be used after all, and creates another contradiction node to force a different choice of room.

N-7	CONTRADICTION	(SL (N-4) ())	
N-8	NOGOOD N-1	(CP N-7 (N-1) ())	*here* ≡ (SL (N-6) ())
N-2	TIME(M) ≠ 1000	(SL (N-8) ())	

Tracing backwards from N-7 through N-4, N-6, and N-1, the backtracker finds that the contradiction depends on only one assumption, N-1. It creates the nogood node N-8, justifies it with a CP-justification, in this case equivalent to the SL-justification (SL (N-6) ()) since N-7's foundations contain N-6 and N-1's repercussions don't. The loss of belief in N-1 carries N-5 away as well, for the TMS makes N-2, N-3, N-6, and N-8 *in*, and N-1, N-4, N-5, and N-7 *out*.

5. Summarizing Arguments

Long or extremely detailed arguments are usually unintelligible. When possible, able expositors make their explanations intelligible by structuring them into clearly separated levels of detail, in which explanations of major points consist of several almost-major points, and so on, with each item explained in terms of the level of detail proper to the item. Structuring arguments in this way serves much the same purpose as structuring plans of action into levels of detail and abstraction. Users of the TMS, or any other means for recording explanations, must take care to convert raw explanations into structured ones if the explanations lack structure initially. Consider, as an exaggerated example, a centralized polling machine for use in national elections. At the end of Election Day, the machine reports that John F. Kennedy has won the election, and when pressed for an explanation of this decision, explains that Kennedy won because Joe Smith voted for him, and Fannie Jones voted for him, and Bert Brown voted for him, *et cetera*, continuing in this way for many millions of voters, pro and con. The desired explanation consists of a summary total of the votes cast for Kennedy, Nixon, and the other candidates. If pressed for further explanations, breakdowns of these totals into state totals follow. The next level of explanation expands into city totals, and only if utterly pressed should the machine break the results into precincts or individual voters for some place, say Cook County.

One can summarize the arguments and explanations recorded in the TMS by using conditional-proofs to subtract nodes representing "low-level" details from arguments. Summarizations can be performed on demand by creating a new node, and justifying this node with a CP-justification mentioning the node to be explained as its consequent, and the set of nodes representing the unwanted low-level beliefs as its *in* and *out*hypotheses. The effective explanation for the new node, via a SL-justification computed from the CP-justification, will consist solely of those high-level nodes present in the well-founded argument being summarized.

Explanations can be generalized or made more abstract by this device as well, by justifying the original node in terms of the new node, or by justifying the original node with the new SL-justification computed for the new node. This new justification will not mention any of the low-level details subtracted from the original argument, and so will support the conclusion as a general result, independent of the particular low-level details used to derive it. For example, an electronic circuit analysis program might compute the voltage gain of an amplifier by assuming a typical input voltage, using this voltage to compute the other circuit voltages including the output voltage, computing the voltage gain as the ratio of the output voltage to the input voltage, and finally justifying the resulting value for the gain using a conditional-proof of the output voltage value given the hypothesized input voltage value. This would leave the gain value depending only on characteristics of the circuit, not on the particular input voltage value used in the computation.

Returning to the election example above, summarizing the election result by subtracting out all the individual voter's ballots leaves the argument empty, for presumably the intermediate results were computed solely in terms of the individual ballots. In order to

summarize an explanation, one must know the form of the desired summarization. While this analytical knowledge is frequently unavoidable, we typically can reduce its scope by introducing structure into arguments during their creation. To illustrate this point, we explain how one simple structuring technique works smoothly with so-called structured descriptions to easily produce perspicuous explanations.

For this discussion, we take a structured description to consist of a description-item (sometimes termed a "node" in the knowledge-representation literature; but we reserve this term for TMS nodes), a set of roles-items representing the parts of the description, and a set of local inference rules. These roles frequently represent entities associated with the description, For example, a PERSON description may have a MOTHER role, and an ADDER description may have roles for ADDEND, AUGEND, and SUM.

To structure explanations, we draw an analogy between the parts of a description and the calling sequence of a procedure. We associate one TMS node with each description-item and two TMS nodes with each role-item. We will use the node associated with a description-item to mark the arguments internal to the description, as explained shortly. We separate the nodes for role-items into two sets, corresponding to the external "calling sequence" and the internal "formal parameters" of the procedure. We use one of the nodes associated with each role-item to represent the external system's view of that "argument" to the procedure call, and the other node associated with the role-item to represent the procedure's view of that "formal parameter" of the procedure. Then we organize the problem solving program so that only the procedure and its internal rules use or justify the internal set of nodes, and that all other descriptions and procedures use or justify only the external set of nodes. The motivation for this separation of the users and justifiers of two sets of nodes into internal users and external users is that we can structure arguments by transmitting information between these two sets of nodes in a special way, as we now describe.

Let D be the node associated with the description, and let E_i and I_i be corresponding external and internal role-item nodes. We justify D to indicate the reason for the validity of the description. To distinguish internal arguments from external arguments, we justify I_i with

(SL (D E_i) ()).

This makes all portions of arguments internal to the description depend on D. With the internal arguments marked in this way, we can separate them from other arguments when transmitting information from the internal nodes to the external nodes. We justify E_i with

(SUM I_i (D) ()).

This justification subtracts all internal arguments from the explanation of E_i, and replaces them with the single node D.

For example, suppose the hypothetical voting program above computed the vote totals for Somewhere, Illinois by summing the totals from the three precincts A, B, and C. The program isolates this level of the computation from the precinct computations by using an ADDER description to sum these subtotals.

N-1	ADDER DESCRIPTION AD	...	*Somewhere, Ill.*
N-2	EXTERNAL A OF AD = 500	...	*Precinct A*
N-3	EXTERNAL B OF AD = 200	...	*Precinct B*
N-4	EXTERNAL C OF AD = 700	...	*Precinct C*

Here the computations for the precinct totals justify the three precinct totals. The program then transmits these values to nodes representing the internal components of the adder description, and a local rule of the description computes the value of the sum component.

N-5	INTERNAL A OF AD = 500	(SL (N-1 N-2) ())	
N-6	INTERNAL B OF AD = 200	(SL (N-1 N-3) ())	
N-7	INTERNAL C OF AD = 700	(SL (N-1 N-4) ())	
N-8	A+B OF AD = 700	(SL (N-5 N-6) ())	*intermediate result*
N-9	INTERNAL SUM OF AD = 1400	(SL (N-7 N-8) ())	

The program transmits this value for the sum to the node representing the external form of the result.

N-10	EXTERNAL SUM OF AC = 1400	(SUM N-9 (N-1) ())	
		here ≡ (SL (N-1 N-2 N-3 N-4) ())	

The SUM-justification for N-10 subtracts all dependence on any internal computations made by the adder, N-1. The resulting explanation for N-10 includes only N-1, N-2, N-3, and N-4. In cases involving more complex procedures than adders, with large numbers of internal nodes and computations of no interest to the external system, the use of this technique for structuring explanations into levels of detail might make the difference between an intelligible explanation and an unintelligible one.

6. Dialectical Arguments

Quine [41] has stressed that we can reject any of our beliefs at the expense of making suitable changes in our other beliefs. For example, we either can change our beliefs to accomodate new observations, or can reject the new observations as hallucinations or mistakes. Notoriously, philosophical arguments have argued almost every philosophical conclusion at the expense of other propositions. Philosophers conduct these arguments in a discipline called dialectial argumentation, in which one argues for a conclusion in two steps; first producing an argument for the conclusion, then producing arguments against the arguments for the 'opposing' conclusion. In this discipline, each debater continually challenges those proposed arguments which he does not like by producing new arguments which either challenge one or more of the premises of the challenged arguments, or which challenge one or more steps of the challenged argument. We can view each debater as following this simplified procedure:

Step 1. *Make an argument*: Put forward an argument A for a conclusion based on premises thought to be shared between the debaters.

Step 2. *Reply to challenges*: When some debater challenges either a premise or a step of A with an argument B, either (1) make a new argument for the conclusion of A, or (2) make an argument for the challenged premise or step of A by challenging one of the premises or steps of B.

Step 3. *Repeat*: Continue to reply to challenges, or make new arguments.

In this section we show how to organize a problem solving program's use of the TMS into the form of dialectical argumentation. Several important advantages and consequences motivate this. As the first consequence, we can reject any belief in a uniform fashion, simply by producing a new, as yet unchallenged, argument against some step or premise of the argument for the belief. We were powerless to do this with the basic TMS mechanisms in any way other than physically removing justifications from the belief system. This ability entails the second consequence, that we must explicitly provide ways to choose what to believe, to select which of the many possible revisions of our beliefs we will take when confronting new information. Quine has urged the fundamentally pragmatic nature of this question, and we must find mechanisms for stating and using pragmatic belief revision rules. As the third consequence of adopting this dialectial program organization, the belief system becomes additive. The system never discards arguments, but accumulates them and uses them whenever possible. This guides future debates by keeping them from repeating past debates. But the arguments alone comprise the belief system, since we derive the current set of beliefs from these arguments. Hence all changes to beliefs occur by adding new arguments to a monotonically growing store of arguments. Finally, as the fourth consequence, the inference system employed by the program becomes modular. We charge each component of the inference system with arguing for its conclusion and against opposing conclusions. On this view, we make each module be a debater rather than an inference rule.

We implement dialectical argumentation in the TMS by representing steps of arguments both by justifications and by beliefs. To allow us to argue against argument steps, we make these beliefs assumptions.

Suppose some module wants to justify node *N* with the justification (SL *I O*). Instead of doing this directly, the module creates a new node, *J*, representing the statement that *I* and *O* SL-justify *N*; in other words, that belief in each node of *I* and lack of belief in each node of *O* constitute a reason for believing in *N*. The module justifies *N* with the justification (SL *J+I O*), where *J+I* represents the list *I* augmented by *J*. The TMS will make *N in* by reason of this justification only if *J* is *in*. The module then creates another new node, $\neg J$, representing the statement that *J* represents a challenged justification. Finally, the module justifies *J* with the justification (SL () ($\neg J$)). In this way, the module makes a new node to represent the justification as an explicit belief, and then assumes that the justification has not been challenged.

For example, suppose a module wishes to conclude that X=3 from X+Y=4 and Y=1. In the dialectial use of the TMS, it proceeds as follows:

N-1	X+Y=4	...
N-2	Y=1	...
N-3	X=3	(SL (N-4 N-1 N-2) ())
N-4	(N-1) AND (N-2) SL-JUSTIFY N-3	(SL () (N-5))
N-5	N-4 IS CHALLENGED	*no justifications yet*

Since N-5 has no justifications, it is *out*, so N-4, and hence N-3, are *in*.

In this discipline, conflicts can be resolved either by challenging premises of arguments, or by challenging those justifications which represent argument steps. Actually, premise justifications for nodes now become assumptions, for the explicit form of the premise justification is itself assumed. In either case, replies to arguments invalidate certain justifications by justifying the nodes representing the challenges. The proponent of the challenged argument can reply by challenging some justification in the challenging argument.

This way of using the TMS clearly makes blind dependency-directed backtracking useless, since the number of assumptions supporting a node becomes very large. Instead, we must use more refined procedures for identifying certain nodes as the causes of inconsistencies. I will return to this issue in Section 8.

7. Models of Other's Beliefs

Many problem solving tasks require us to reason about the beliefs of some agent. For example, according to the speech act theory of purposeful communication, I must reason about your beliefs and wants, and about your beliefs about my beliefs and wants. [5, 49] This requirement entails the ability to reason about embedded belief and want spaces, so called because these are sets of beliefs reported as compound or embedded statements of belief, for example, "I believe that you believe that it is raining." In fact, I must frequently reason about my own system of beliefs and wants. In planning, I must determine what I will believe and want after performing some actions, and this requires my determining how the actions affect my beliefs and wants. In explaining my actions, I must determine what I believed and wanted at some time in the past, before performing the intervening actions. And when choosing what to do, want or believe, I must reason about what I now am doing, wanting, or believing.

In making a problem solver which uses such models of belief systems, we face the problem of how to describe how additions to these models affect the beliefs contained in them, how belief revision proceeds within a model of a belief system. It would be extremely convenient if the same mechanism by which the program revises its own beliefs, namely the TMS, could be applied to revising these models as well. Fortuitously, the mechanism of representing justifications as explicit beliefs introduced in Section 6 lets us do just that.

To represent a belief system within our own, we use beliefs in our own system about the beliefs and justifications in the other system. We then mirror the other system's justifications of its beliefs by making corresponding justifications in our system's TMS of our beliefs about the other system's beliefs. For each node *N* in an agent U's belief system,

we make two nodes, UB[*N*] and ¬UB[*N*], one representing that U believes in *N*, and the other representing that U doesn't believe in *N*. We justify ¬UB[*N*] with (SL () (UB[*N*])), thus assuming that all nodes in U's system are *out* until given valid justifications. For each SL-justification *J* (= (SL *I O*)) for *N* in U's belief system, we make a node UB[*J*] representing that U's belief system contains *J*. We then justify UB[*N*] with the justification (SL *L* ()), where *L* contains the node UB[*J*], the nodes UB[*M*] for each node *M* in *I*, and the nodes ¬UB[*M*] for each node *M* in *O*.

We might view this technique as embodying an observation of traditional modal logics of belief. Most of these logics include an axiom schema about the belief modality Bel of the form

$$Bel(p \supset q) \supset (Bel(p) \supset Bel(q)).$$

When we mirror embedded justifications with TMS justifications, we are making an inference analogous to the one licensed by this axiom.

For example, suppose the program believes that an agent U believes A, that U does not believe B, and that U believes C because he believes A and doesn't believe B. If we looked into U's TMS we might see the following nodes and justifications.

A	...	...
B	...	...
C	...	(SL (A) (B))

Our program represents this fragment of U's belief system as follows:

N-1	UB[A]	...	*UB means "U believes"*
N-2	¬UB[B]	(SL () (N-3))	
N-3	UB[B]		*no justification yet*
N-4	UB[(A) AND (B) SL-JUSTIFY C]	...	
N-5	UB[C]	(SL (N-4 N-1 N-2) ())	

In this case, N-1, N-2, N-4, and N-5 are *in*, and N-3 is *out*. If the program revises its beliefs so that N-2 is *out*, say by the addition of a new justification in U's belief system,

N-5	UB[... SL-JUSTIFY B]	...
N-2	UB[B]	(SL (N-5 ...) ())

then the TMS will make N-5 *out* as well. In this way, changes made in particular beliefs in the belief system lead automatically to other changes in beliefs which represent the implied changes occurring within the belief system.

Of course, we can repeat this technique at each level of embedded belief spaces. To use this technique we must require the inference rules which draw conclusions inside a belief space to assert the corresponding beliefs about justifications for those conclusions.

One frequently assumes that others have the same set of inference rules as oneself, that they can draw the same set of inferences from the same set of hypotheses. This assumption amounts to that of assuming that everyone operates under the same basic "program" but has different data or initial beliefs. I am currently exploring a formalization of this assumption in an "introspective" problem solver which has a description of itself as a program, and which uses this self-description as the basis of its models of the behavior of others. [16]

8. Assumptions and the Problem of Control

How a problem solver revises its beliefs influences how it acts. Problem solvers typically revise their beliefs when new information (such as the expected effect of an action just taken or an observation just made) contradicts previous beliefs. These inconsistencies may be met by rejecting the belief that the action occurred or that the observation occurred. This might be thought of as the program deciding it was hallucinating. Sometimes, however, we choose to reject the previous belief and say that the action made a change in the world, or that we had made some inappropriate assumption which was corrected by observation. Either of these ways of revising beliefs may be warranted in different circumstances. For example, if during planning we encounter a contradiction by thinking through a proposed sequence of actions, we might decide to reject one of the proposed actions and try another action. On the other hand, if while carrying out a sequence of actions we encounter a contradiction in our beliefs, we might decide that some assumption we had about the world was wrong, rather than believe that we never took the last action. As this example suggests, we might choose to revise our beliefs in several different ways. Since we decide what to do based on what we believe and what we want, our choice of what to believe affect what we choose to do.

How can we guide the problem solver in its choice of what to believe? It must make its choice by approximating the set of possible revisions by the set of assumptions it can change directly, for it cannot see beforehand all the consequences of a change without actually making that change and seeing what happens. We have studied two means by which the problem solver can decide what to believe, the technique of encoding control information into the set of justifications for beliefs, and the technique of using explicit choice rules. Both of these approaches amount to having the reasoner deliberate about what to do. In the first case, the reasoning is "canned." In the second, the reasoning is performed on demand.

We encode some control information into the set of justifications for beliefs by using patterns of non-monotonic justifications. We can think of a non-monotonic justification (SL () (N-2 N-3)) for N-1 as suggesting the order in which these nodes should be believed, N-1 first, then N-2 or N-3 second. On this view, each non-monotonic justification contributes a fragment of control information which guides how the problem solver revises its beliefs. In Sections 8.1, 8.2, and 8.3 we illustrate how to encode several standard control structures in patterns of justifications, namely default assumptions, sequences of alternatives, and a way of choosing representatives of equivalence classes useful

in controlling propagation of constraints (a deduction technique presented by [53]). These examples should suggest how other control structures such as decision trees or graphs might be encoded.

Even with these fragments of control information, many alternative revisions may appear possible to the problem solver. In such cases, we may wish to provide the problem solver with rules or advice about how to choose which revision to make. If we are clever (or lazy), we might structure the problem solver so that it uses the same language and mechanisms for these revision rules as for rules for making other choices, such as what action to perform next, how to carry out an action, or which goal to pursue next. In [11] my colleagues and I incorporated this suggestion into a general methodology which we call *explicit control of reasoning*, and implemented AMORD, a language of pattern-invoked procedures controlled by the TMS. I am currently studying a problem solver architecture, called a *reflexive interpreter*, in which the problem solver's structure and behavior are themselves domains for reasoning and action by the problem solver [16]. This sort of interpreter represents its own control state to itself explicitly among its beliefs as a *task network* similar to that used in McDermott's [34] NASL, in which problems or intentions are represented as *tasks*. The interpreter also represents to itself its own structure as a program by means of a set of *plans*, abstract fragments of task network. It represents the important control state of having to make a choice by creating a *choice task*, whose carrying out involves making a choice. The interpreter can then treat this choice task as a problem for solution like any other task. In this framework, we formulate rules for guiding belief revision as plans for carrying out choice tasks. We index these revision plans by aspects of the problem solver state; for example, by the historical state, by the control state, by the state of the problem solution, by the domain, by the action just executed, and by other circumstances. Each revision plan might be viewed as a specialization of the general dependency-directed backtracking procedure. Such refinements of the general backtracking procedure take the form of the beliefs (and thus the problem solver state) into account when deciding which assumptions should be rejected. I will report the details of my investigation in my forthcoming thesis.

8.1 Default Assumptions

Problem solving programs frequently make specifications of default values for the quantities they manipulate, with the intention either of allowing specific reasons for using other values to override the current values, or of rejecting the default if it leads to an inconsistency. (See [45] for a lucid exposition of some applications.) The example in Section 1.2 includes such a default assumption for the day of the week of a meeting.

To pick the default value from only two alternatives, we justify the default node non-monotonically on the grounds that the alternative node is *out*. We generalize this binary case to choose a default from a larger set of alternatives. Take $S = \{A_1, ..., A_n\}$ to be the set of alternative nodes, and if desired, let G be a node which represents the reason for making an assumption to choose the default. To make A_i the default, justify it with

(SL (*G*) (A_1 ... A_{i-1} A_{i+1} ... A_n)).

If no additional information about the value exists, none of the alternative nodes except A_i will have a valid justification, so A_i will be *in* and each of the other alternative nodes will be *out*. Adding a valid justification to some other alternative node causes that alternative to become *in*, and invalidates the support of A_i, so A_i goes *out*. When analyzing a contradiction derived from A_i, the dependency-directed backtracking mechanism recognizes A_i as an assumption because it depends on the other alternative nodes being *out*. The backtracker may then justify one of the other alternative nodes, say A_j, causing A_i to go *out*. This backtracker-produced justification for A_j will have the form

(SL <various nodes> <remainder nodes>)

where <remainder nodes> is the set of A_k's remaining in S after A_i and A_j are taken away. In effect, the backtracker removes the default node from the set of alternatives, and makes a new default assumption from the remaining alternatives. As a concrete example, our scheduling program might default a meeting day as follows:

N-1	PREFER W. TO M. OR F.	...
N-2	DAY(M) = MONDAY	
N-3	DAY(M) = WEDNESDAY	(SL (N-1) (N-2 N-4))
N-4	DAY(M) = FRIDAY	

The program assumes Wednesday to be the day of the meeting M, with Monday and Friday as alternatives. The TMS will make Wednesday the chosen day until the program gives a valid reason for taking Monday or Friday instead

We use a slightly different set of justifications if the complete set of alternatives cannot be known in advance but must be discovered piecemeal. This ability to extend the set of alternatives is necessary, for example, when the default is a number, due to the large set of possible alternatives. Retaining the above notation, we represent the negation of St(A_i) with a new node, $\neg A_i$. We arrange for A_i to be believed if $\neg A_i$ is *out*, and set up justifications so that if A_j is distinct from A_i, A_j supports $\neg A_i$. We justify A_i with

(SL (*G*) ($\neg A_i$)),

and justify $\neg A_i$ with a justification of the form

(SL (A_j) ())

for each alternative A_j distinct from A_i. As before, A_i will be assumed if no reasons for using any other alternative exist. Furthermore, new alternatives can be added to the set S simply by giving $\neg A_i$ a new justification corresponding to the new alternative. As before, if the problem solving program justifies an unselected alternative, the TMS will make the default node *out*. Backtracking, however, has a new effect. If A_i supports a contradiction, the backtracker may justify $\neg A_i$ so as to make A_i become *out*. When this happens, the TMS has no way to select an alternative to take the place of the default assumption. The extensible structure requires an external mechanism to construct a new default assumption

whenever the current default is ruled out. For example, a family planning program might make assumptions about the number of children in a family as follows:

N-1	PREFER 2 CHILDREN	...
N-2	#-CHILDREN(F) = 2	(SL (N-1) (N-3))
N-3	#-CHILDREN(F) ≠ 2	(SL (N-4) ())
N-4	#-CHILDREN(F) = 0	(SL (N-5) ())
N-5	#-CHILDREN(F) = 1	(SL (N-6) ())
N-6	#-CHILDREN(F) = 3	(SL (N-7) ())
N-7	#-CHILDREN(F) = 4	

With this system of justifications, the TMS would make N-2 *in*. If the planning program finds some compelling reason for having 5 children, it would have to create a new node to represent this fact, along with a new justification for N-3 in terms of this new node.

8.2 Sequences of Alternatives

Linearly ordered sets of alternatives add still more control information to a default assumption structure, namely the order in which the alternatives should be tried. This extra heuristic information might be used, for example, to order selections of the day of the week for a meeting, of a planning strategy, or of the state of a transistor in a proposed circuit analysis.

We represent a sequence of alternatives by a controlled progression of default assumptions. Take $\{A_1, ..., A_n\}$ to be the heuristically-ordered sequence of alternative nodes, and let G be a node which represents the reason for this heuristic ordering. We justify each A_i with

$$(\text{SL } (G \neg A_{i-1}) (\neg A_i)).$$

A_1 will be selected initially, and as the problem solver rejects successive alternatives by justifying their negations, the TMS will believe the successive alternatives in turn. For example, our scheduling program might have:

N-1	SEQUENCE N-2 N-4 N-6	...
N-2	DAY(M) = WEDNESDAY	(SL (N-1) (N-3))
N-3	DAY(M) ≠ WEDNESDAY	
N-4	DAY(M) = THURSDAY	(SL (N-1 N-3) (N-5))
N-5	DAY(M) ≠ THURSDAY	
N-6	DAY(M) = TUESDAY	(SL (N-1 N-5) ())

This would guide the choice of day for the meeting M to Wednesday, Thursday and

Tuesday, in that order.

Note that this way of sequencing through alternatives allows no direct way for the problem solving program to reconsider previously rejected alternatives. If, say, we wish to use special case rules to correct imprudent choices of culprits made by the backtracking system, we need a more complicated structure to represent linearly ordered alternatives. We create three new nodes for each alternative A_i: PA_i, which means that A_i is a possible alternative, NSA_i, which means that A_i is not the currently selected alternative, and ROA_i, which means that A_i is a ruled-out alternative. We suggest members for the set of alternatives by justifying each PA_i with the reason for including A_i in the set of alternatives. We leave each ROA_i unjustified, and justify each A_i and NSA_i with

A_i: (SL (PA_i NSA_1 ... NSA_{i-1}) (ROA_i))
NSA_i: (SL () (PA_i))
(SL (ROA_i) ()).

Here the justification for A_i is valid if and only if A_i is an alternative, no better alternative is currently selected, and A_i is not ruled out. The two justifications for NSA_i mean that either A_i is not a valid alternative, or that A_i is ruled out. With this structure, different parts of the problem solver can independently rule in or rule out an alternative by justifying the appropriate A or ROA node. In addition, we can add new alternatives to the end of such a linear order by constructing justifications as specified above for the new nodes representing the new alternative.

8.3 Equivalence Class Representatives

Problem solvers organized to encourage modularity and additivity frequently contain several different methods or rules which compute values for the same quantity or descriptions for the same object. We call these multiple results *coincidences*, after [9, 59]. If the several methods compute several values, we can often derive valuable information by checking these competing values for consistency. With polynomials as values, for example, this consistency checking sometimes allows solving for the values of one or more variables. After checking the coincidence for consistency and new information, prudent programs normally use only one of the suggested values in further computation, and retain the other values for future reference. The various values form an *equivalence class* with respect to the propagation of the value in other computations, and one of the values in this class must be chosen as the representative for propagation. We could choose the representative with a default assumption, but this would lead to undesirable backtracking behavior. For instance, if the backtracking system finds the equivalence class representative involved in an inconsistency, then it should find some way of rejecting the representative as a proper value, rather than letting it stand and selecting a new representative. This means the backtracker should find the choice of representative invisible, and this requirement rules out using

either the default assumption or sequence of alternatives representations.

We select equivalence class representatives by using conditional-proof justifications to hide the choice mechanism from the backtracking system. For each node R_i representing a equivalence class member, we create two new nodes: PR_i, which means that R_i is a possible representative, and SR_i, which means that R_i is the selected representative. Rather than the program deriving justifications for the R nodes directly, it should instead suggest these values as possible representatives by justifying the corresponding PR nodes instead. With this stipulation we justify each R_i and SR_i as follows:

SR_i: (SL (PR_i) (SR_1 ... SR_{i-1}))
R_i: (CP SR_i () (SR_1 ... SR_{i-1}))

Here the justification for SR_i means that R_i has been suggested as a member of the equivalence class, and that no other member of the class has been selected as the representative. This justification constitutes a default assumption of R_i as the selected representative. However, the justification for R_i means that the reason for believing R_i will be the reason for both suggesting and selecting R_i (the argument for SR_i), minus the reason for selecting it (the default assumption), thus leaving only the reason for which R_i was suggested, namely the antecedents of PR_i. In this way the equivalence class selector picks alternatives as the representative in the order in which they were added to the set of alternatives, while hiding this selection from the backtracking system.

For example, suppose a commodity analysis program derives two values for the predicted number of tons of wheat grown this year and notices this coincidence.

```
N-1     SUGGEST WHEAT(1979) = 5X+3000        (SL (R-57 ...) ())
N-2     SUGGEST WHEAT(1979) = 7Y             (SL (R-60 ...) ())
N-3     Y = (5X+3000)/7                      (SL (N-1 N-2) ())
```

These suggested values correspond to possible equivalence class representatives. To avoid using both values in further computations, the program chooses one.

```
N-4     N-1 SELECTED                         (SL (N-1) ())
N-5     WHEAT(1979) = 5X+3000                (CP N-4 () ())
                                             here ≡ (SL (N-1) ())
N-6     N-2 SELECTED                         (SL (N-2) (N-4))
N-7     WHEAT(1979) = 7Y                     (CP N-6 () (N-4))
```

Since N-1 is *in* and it is the first in the ordering imposed by the selection justifications, it is selected to be the value propagated, and the TMS makes N-4 and N-5 *in*. Suppose now that some contradiction occurs and has N-5 in its foundations, and that the dependency-directed backtracker denies some assumption in N-1's foundations.

Consequently, the TMS makes N-1 (and N-3) *out*, and N-6 and N-7 *in*, with N-7's CP-justification equivalent in this case to (SL (N-2) ()). Thus the pattern of justifications leads to selecting the second suggested value. If the program then finds a third value,

```
N-8     SUGGEST WHEAT(1979) = 4X+100
N-9     Y = (4X+100)/7
N-10    N-8 SELECTED
N-11    WHEAT(1979) = 4X+100
```

```
...
(SL (N-2 N-8) ()
(SL (N-8) (N-4 N-6))
(CP N-10 () (N-4 N-6))
```

it will derive any new information possible from the new value, then add the new value to the list of waiting alternative values for propagation. In this case, N-8 and N-9 are *in*, and N-10 and N-11 are *out*.

9. Experience and Extensions

We have experience with the TMS in a number of programs and applications. I implemented the first version of the TMS in September 1976, as an extension (in several ways) of the "fact garbage collector" of Stallman and Sussman's [53] ARS electronic circuit analysis program. After that, I took the program through many different versions. Howard Shrobe and James Stansfield also made improvements in the program. Truth maintenance techniques have been applied in several other circuit analysis and synthesis programs, including SYN [12] and QUAL [10], and in Steele and Sussman's [56] constraint language. We organized our rule-based problem solving system AMORD [11] around the TMS, and used AMORD in developing a large number of experimental programs, ranging from blocks world problem solvers to circuit analysis programs and compilers. McAllester [31] uses his own truth maintenance system in a program for symbolic algebra, electronic circuit analysis, and programming. In addition, Weiner [62] of UCLA used AMORD to implement an explanation system in studying the structure of natural explanations, and Shrobe [50] uses AMORD in a program-understanding system.

Several researchers have extended the basic belief-revision techniques of the TMS by embedding them in larger frameworks which incorporate time, certainty measures, and other problem solving concepts and processes. Friedman [19] and Stansfield [54] merge representations of continuous degrees of belief with truth maintenance techniques. London [29] and Thompson [60] add chronological contexts to a dependency-based framework. London also presents many detailed examples of the use of dependency networks in the modelling component of a problem solving program.

Improvements in the basic truth maintenance process have also been suggested. McAllester [30] describes a relative of the TMS based on a three-valued belief set, with multi-directional clauses as justifications. Thompson [60] generalizes the node-justification structure of the TMS to non-clausal arguments and justifications. Shrobe (in [50] and in personal communications) has suggested several ways in which the problem solver can profit by reasoning about the structure of arguments, particularly in revising its set of goals

after solving some particular goal. These ideas suggest other improvements including (1) modification of the TMS to make use of multiple supporting-justifications whenever several well-founded arguments can be found for a node, (2) use of the TMS to signal the problem solver whenever the argument for some node changes, and (3) development of a language for describing and efficiently recognizing patterns in arguments for nodes, well-founded or otherwise. How to incorporate truth maintenance techniques into "virtual-copy" representational systems [17] also seems worth study.

The TMS continually checks CP-justifications for validity, in hopes of deriving new equivalent SL-justifications. This makes the implementation considerably more complex than one might imagine. I expect that a simpler facility would be more generally useful, namely the TMS without CP-justifications, but with the Find Independent Support procedure isolated as a separate, user invoked facility. Practical experience shows that in most cases, one only expects the CP-justification to be valid in the situation in which it is created, so it seems reasonable to make the user responsible for calling FIS directly rather than letting the TMS do it.

Finally, a cluster of problems center about incrementality. The TMS normally avoids examining the entire data base when revising beliefs, and instead examines only the repercussions of the changed nodes. However, apparently unsatisfiable circularities can occur which require examining nodes not included in these repercussions. In another sense of incrementality, some circumstances can force the TMS to examine large numbers of nodes, only to leave most of them in their original state after finding alternate non-circular arguments for the supposedly changed nodes. Latombe [27] has investigated ways of avoiding this, but these difficulties deserve further study. One particularly enticing possibility is that of adapting the ideas of Baker's [1] real-time list garbage-collection algorithms to the case of truth maintenance.

10. Discussion

The TMS solves part of the belief revision problem, and provides a mechanism for making non-monotonic assumptions. Artificial intelligence researchers recognized early on that AI systems must make assumptions, and many of their systems employed some mechanism for this purpose. Unfortunately, the related problem of belief revision received somewhat less study. Hayes [21] emphasised the importance of the belief revision problem, but with the exception of Colby [6], who employed a belief system with reasons for some beliefs, as well as measures of credibility and emotional importance for beliefs, most work on revising beliefs appears to have been restricted to the study of backtracking algorithms operating on rather simple systems of states and actions. The more general problem of revising beliefs based on records of inferences has only been examined in more recent work, including Cox's [7] graphical deduction system, Crocker's [8] verification system, de Kleer's [9] electronic circuit analysis program, Fikes' [18] deductive modelling system, Hayes' [22] travel planning system, Katz and Manna's [24] program modification system, Latombe's [26, 27] design program, London's [29] planning and modelling system, McDermott's [32, 33, 34] language understanding, data base, and design programs, Moriconi's [39] verification system, Nevins'

[40] theorem prover, Shrobe's [50] program understanding system, the MDS/AIMDS/BELIEVER programs [47, 51, 52], and Sussman and Stallman's [53, 59] electronic circuit analysis programs. Berliner's [3] chess program employed "lemmas" for recording interesting facts about partial board analyses reminiscient of conditional-proofs, but the program derives these lemmas through a perturbation technique rather than through analysis of arguments and justifications.

In addition, the philosophical literature includes many treatments of belief revision and related problems. Many writers study evaluative criteria for judging which belief revisions are best, based on the connections between beliefs. Quine and Ullian [43] survey this area. Other writers study the problems of explanations, laws, and counterfactual conditionals. Rescher [44] builds on Goodman's [20] exposition of these problems to present a framework for belief revision motivated by Quine's [41, 42] "minimum mutilation" principle. Lewis [28] and Turner [61] propose means for filling in this framework. Scriven [48] relates these questions to the problem of historical explanation in a way quite reminiscient of our non-monotonic arguments for beliefs. Suppes [57] surveys work on learning and rational changes of belief. The view of reasoning proposed in Section 1.1 is connected with many topics in the theories of belief, action, and practical reasoning. Minsky [37] presents a theory of memory which includes a more general view of reasoning.

Kramosil [25] initiated the mathematical study of non-monotonic inference rules, but reached pessimistic conclusions. More recently, McDermott and I [35] attempt to formalize the logic underlying the TMS with what we call *non-monotonic logic*. We also survey the history of such reasoning techniques. Weyhrauch [63] presents a framework for meta-theoretic reasoning in which these reasoning techniques and others can be expressed. Hintikka [23] presents a form of possible-world semantics for modal logics of knowledge and belief. Moore [38] combines this semantics for knowledge with a modal logic of action, but ignores belief and belief revision.

One might hope to find clues about how to organize the TMS's analysis of potential arguments for beliefs by studying what types of arguments humans find easy or difficult to understand. Statman [55] indicates that humans have difficulty following arguments which have many back-references to distant statements. He attempts to formalize some notions of the complexity of proofs using measures based on the topology of the proof graph. Wiener [62] catalogues and analyzes a corpus of human explanations, and finds that most exhibit a fairly simple structure. De Kleer [10] studies causal explanations of the sort produced by engineers, and discovers that a few simple principles govern a large number of these explanations.

I have used the term "belief" freely in this paper, so much so that one might think the title "Belief Revision System" more appropriate, if no less ambitious, than "Truth Maintenance System." Belief, however, for many people carries with it a concept of grading, yet the TMS has no non-trivial grading of beliefs. (Section 9 mentioned some extensions which do.) Perhaps a more accurate label would be "opinion revision system," where I follow Dennett [13] in distinguishing between binary judgemental assertions (opinions) and graded underlying feelings (beliefs). As Dennett explains, this distinction permits description of those circumstances in which reasoned arguments force one to assert a

conclusion, even though one does not believe the conclusion. Hesitation, self-deception, and other complex states of belief and opinion can be described in this way. I feel it particularly apt to characterize the TMS as revising opinions rather than beliefs. Choosing what to "believe" in the TMS involves making judgements, rather than continuously accreting strengths or confidences. A single new piece of information may lead to sizable changes in the set of opinions, where new beliefs typically change old ones only slightly.

I also find this distinction between binary judgements and graded approximations useful in distinguishing non-monotonic reasoning from imprecise reasoning, such as that modelled by Zadeh's [64] fuzzy logic. I view the non-monotonic capabilities of the TMS as capabilities for dealing with incomplete information, but here the incompleteness is "exact": it makes binary statements about (typically) precise statements. Any approximation in the logic enters only when one views the set of current beliefs as a whole. In the logics of imprecise reasoning, the incompleteness is "inexact": the statements themselves are vague, and the vagueness need not be a property of the entire system of beliefs. While both approaches appear to be concerned with related issues, they seem to be orthogonal in their current development, which suggests studies of their combination. (Cf. [19].)

One final note: The overhead required to record justifications for every program belief might seem excessive. Some of this burden might be eliminated by using the summarization techniques of Section 5 to replace certain arguments with smaller ones, or by adopting some (hopefully well-understood) discipline of retaining only essential records from which all discarded information can be easily recomputed. However, the pressing issue is not the expense of keeping records of the sources of beliefs. Rather, we must consider the expense of *not* keeping these records. If we throw away information about derivations, we may be condemning ourselves to continually rederiving information in large searches caused by changing irrelevant assumptions. This original criticism of MICRO-PLANNER (in [58]) applies to the context mechanisms of CONNIVER and QA4 as well. If we discard the sources of beliefs, we may make impossible the correction of errors in large, evolving data bases. We will find such techniques not just desirable, but necessary, when we attempt to build truly complex programs and systems. Lest we follow the tradition of huge, incomprehensible systems which spawned Software Engineering, we must, in Gerald Sussman's term, make "responsible" programs which can explain their actions and conclusions to a user. (Cf. [46].)

References

[1] Baker, H. G. Jr., "List Processing in Real Time on a Serial Computer," *Comm. ACM*, Vol. 21, No. 4, (April 1978), pp. 280-294.

[2] Belnap, N. D., "How a Computer Should Think," in *Contemporary Aspects of Philosophy*, Gilbert Ryle, ed., Oriel Press, Stocksfield, 1976.

[3] Berliner, H. J., "Chess as Problem Solving: The Development of a Tactics Analyzer," CMU Computer Science Department, 1974.

[4] Charniak, E.,Riesbeck, C., and McDermott, D., *Artificial Intelligence Programming*, Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1979.

[5] Cohen, P. R., "On Knowing What to Say: Planning Speech Acts," Department of Computer Science, University of Toronto, TR-118, 1978.

[6] Colby, K. M., "Simulations of Belief Systems," in R. C. Schank and K. M. Colby, editors, *Computer Models of Thought and Language*, W. H. Freeman, San Francisco, 1973, pp. 251-286.

[7] Cox, P. T., "Deduction Plans: A Graphical Proof Procedure for the First-order Predicate Calculus," Department of Computer Science, University of Waterloo, Research Report CS-77-28, 1977.

[8] Crocker, S. D., "State Deltas: A Formalism for Representing Segments of Computation," University of Southern California, Information Sciences Institute, RR-77-61, 1977.

[9] de Kleer, J., "Local Methods for Localization of Failures in Electronic Circuits," MIT AI Lab, Memo 394, November 1976.

[10] de Kleer, J., "Causal and Teleological Reasoning in Circuit Recognition," Ph.D. thesis, MIT Department of Electrical Engineering and Computer Science, 1979.

[11] de Kleer, J., Doyle, J., Steele, G. L Jr. and Sussman, G. J., "Explicit Control of Reasoning," *Proc. ACM Symp. on Artificial Intelligence and Programming Languages*, Rochester, New York, 1977, also MIT AI Lab, Memo 427, 1977.

[12] de Kleer, J. and Sussman, G. J., "Propagation of Constraints Applied to Circuit Synthesis," MIT AI Lab, Memo 485, 1978.

[13] Dennett, D. C., "How to Change Your Mind," in *Brainstorms*, Bradford Books, Montgomery, Vermont, 1978, pp. 300-309.

[14] Doyle, J., "The Use of Dependency Relationships in the Control of Reasoning," MIT AI Lab, Working Paper 133, 1976.

[15] Doyle, J., "Truth Maintenance Systems for Problem Solving," MIT AI Lab, TR-419, 1978a.

[16] Doyle, J., "Reflexive Interpreters", MIT Department of Electrical Engineering and Computer Science, Ph.D. proposal, 1978b.

[17] Fahlman, S. E., *NETL: A System for Representing and Using Real World Knowledge*, MIT Press, Cambridge, 1979.

[18] Fikes, R. E., "Deductive Retrieval Mechanisms for State Description Models," *Proc. Fourth International Joint Conference on Artificial Intelligence*, 1975, pp. 99-106.

[19] Friedman, L., "Plausible Inference: A Multi-Valued Logic for Problem Solving," Jet Propulsion Laboratory, Pasadena, California, Report 79-11, 1979.

[20] Goodman, N., "The Problem of Counterfactual Conditionals," in *Fact, Fiction, and Forecast*, third edition, Bobbs-Merrill, New York, 1973, pp. 3-27.

[21] Hayes, P. J., "The Frame Problem and Related Problems in Artificial Intelligence," in A. Elithorn and D. Jones, editors, *Artificial and Human Thinking*, Josey-Bass, San Francisco, 1973.

[22] Hayes, P. J., "A Representation for Robot Plans," *Proc. Fourth IJCAI*, 1975, pp. 181-188.

[23] Hintikka, J., *Knowledge and Belief*, Cornell University Press, Ithica, 1962.

[24] Katz, S. and Manna, Z., "Logical Analysis of Programs," *Comm. ACM*, Vol. 19, No. 4, (1976), pp. 188-206.

[25] Kramosil, I., "A Note on Deduction Rules with Negative Premises," *Proc. Forth International Joint Conference on Artificial Intelligence*, 1975, pp. 53-56.

[26] Latombe, J.-C., "Une Application de l'intelligence Artificielle a la Conception Assistee par Ordinateur (TROPIC)," Universite Scientifique et Medicale de Grenoble, thesis D.Sc. Mathematiques,.1977.

[27] Latombe, J.-C., "Failure Processing in a System for Designing Complex Assemblies," submitted to IJCAI, 1979.

[28] Lewis, D., *Counterfactuals*, Basil Blackwell, London, 1973.

[29] London, P. E., "Dependency Networks as a Representation for Modelling in General Problem Solvers," Department of Computer Science, University of Maryland, TR-698, 1978.

[30] McAllester, D. A., "A Three-Valued Truth Maintenance System", MIT AI Lab, Memo 473, 1978.

[31] McAllester, D. A., "The Use of Equality in Deduction and Knowledge Representation," MIT Department of Electrical Engineering and Computer Science, M.S. thesis, 1979.

[32] McDermott, D., "Assimilation of New Information by a Natural Language-Understanding System," MIT AI Lab, AI-TR-291, 1974.

[33] McDermott, D., "Very Large PLANNER-type Data Bases," MIT AI Lab, AI Memo 339, 1975.

[34] McDermott, D., "Planning and Acting," *Cognitive Science*, Vol. 2, (1978), pp. 71-109.

[35] McDermott, D. and Doyle, J., "Non-Monotonic Logic I", MIT AI Lab, Memo 486, 1978.

[36] Minsky, M., "A Framework for Representing Knowledge," MIT AI Lab, Memo 306, 1974.

[37] Minsky, M., "K-lines: A Theory of Memory," MIT AI Lab, Memo 516, 1979.

[38] Moore, R. C., "Reasoning About Knowledge and Action," Ph.D. thesis, MIT Department of Electrical Engineering and Computer Science, 1979.

[39] Moriconi, M. "A System for Incrementally Designing and Verifying Programs," University of Southern California, Information Sciences Institute, RR-77-65, 1977.

[40] Nevins, A. J., "A Human-Oriented Logic for Automatic Theorem Proving," *J. ACM 21*, #4 (October 1974), pp. 606-621.

[41] Quine, W. V., "Two Dogmas of Empiricism," in *From a Logical Point of View*, Harvard University Press, Cambridge, 1953.

[42] Quine, W. V., *Philosophy of Logic*, Prentice-Hall, Englewood Cliffs, 1970.

[43] Quine, W. V. and Ullian, J. S., *The Web of Belief*, second edition, Random House, New York, 1978.

[44] Rescher, N., *Hypothetical Reasoning*, North Holland, Amsterdam, 1964.

[45] Reiter, R., "On Reasoning by Default," *Proc. Second Symp. on Theoretical Issues in Natural Language Processing*, Urbana, Illinois, 1978.

[46] Rich, C., Shrobe, H. E., and Waters, R. C., "Computer Aided Evolutionary Design for Software Engineering," MIT AI Lab, AI Memo 506, 1979.

[47] Schmidt, C. F. and Sridharan, N. S., "Plan Recognition Using a Hypothesize and Revise Paradigm: an Example," *Proc. Fifth IJCAI*, 1977, pp. 480-486.

[48] Scriven, M., "Truisms as the Grounds for Historical Explanations," in P. Gardiner, ed., *Theories of History*, Free Press, New York, 1959.

[49] Searle, J. R., *Speech Acts*, Cambridge University Press, 1969.

[50] Shrobe, H. E., "Dependency Directed Reasoning for Complex Program Understanding," MIT AI Lab, TR-503, 1979.

[51] Sridharan, N. S. and Hawrusik, F., "Representation of Actions that have Side-Effects," *Proc. Fifth IJCAI*, 1977, pp. 265-266.

[52] Srinivasan, C. V., "The Architecture of Coherent Information System: A General Problem Solving System," *IEEE Transactions on Computers*, Vol. C-25, No. 4, April 1976, pp. 390-402.

[53] Stallman, R. M. and Sussman, G. J., "Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis," *Artificial Intelligence*, Vol. 9, No. 2, (October 1977), pp. 135-196.

[54] Stansfield, J. L., "Integrating Truth Maintenance Systems with Propagation of Strength of Belief as a Means of Addressing the Fusion Problem," MIT AI Lab, draft proposal, 1978.

[55] Statman, R., "Structural Complexity of Proofs," Stanford University, Department of Mathematics, Ph.D. thesis, 1974.

[56] Steele, G. L. Jr. and Sussman, G. J., "Constraints," MIT AI Lab, Memo 502, 1978.

[57] Suppes, P., "A Survey of Contemporary Learning Theories," in R. E. Butts and J. Hintikka, editors, *Foundational Problems in the Special Sciences*, D. Reidel, Dordrecht, 1977.

[58] Sussman, G. J. and McDermott, D., "From PLANNER to CONNIVER - A genetic approach," *Proc. AFIPS FJCC*, 1972, pp. 1171-1179.

[59] Sussman, G. J. and Stallman, R. M., "Heuristic Techniques in Computer-Aided Circuit Analysis," *IEEE Transactions on Circuits and Systems*, Vol. CAS-22, No. 11, (November 1975), pp. 857-865.

[60] Thompson, A., "Network Truth Maintenance for Deduction and Modelling," *Proc. Sixth IJCAI*, 1979.

[61] Turner, R., "Counterfactuals Without Possible Worlds," University of Essex, 1978.
[62] Weiner, J., "The Structure of Natural Explanations: Theory and Applications," Ph.D. thesis, University of California, Los Angeles, 1979.
[63] Weyhrauch, R. W., "Prolegomena to a Theory of Formal Reasoning," Stanford AI Lab, AIM-315, 1978.
[64] Zadeh, L., "Fuzzy Logic and Approximate Reasoning," *Synthese* 30, (1975), pp. 407-428.

Node	Justification	Justification Name
1	(SL (3) ())	J1
2	(SL () (1))	J2
3	(SL (1) ())	J3
4	(SL (2) ())	J4a
4	(SL (3) ())	J4b
5	(SL () ())	J5
6	(SL (3 5) ())	J6

Figure 1. A sample system of six nodes and seven justifications.

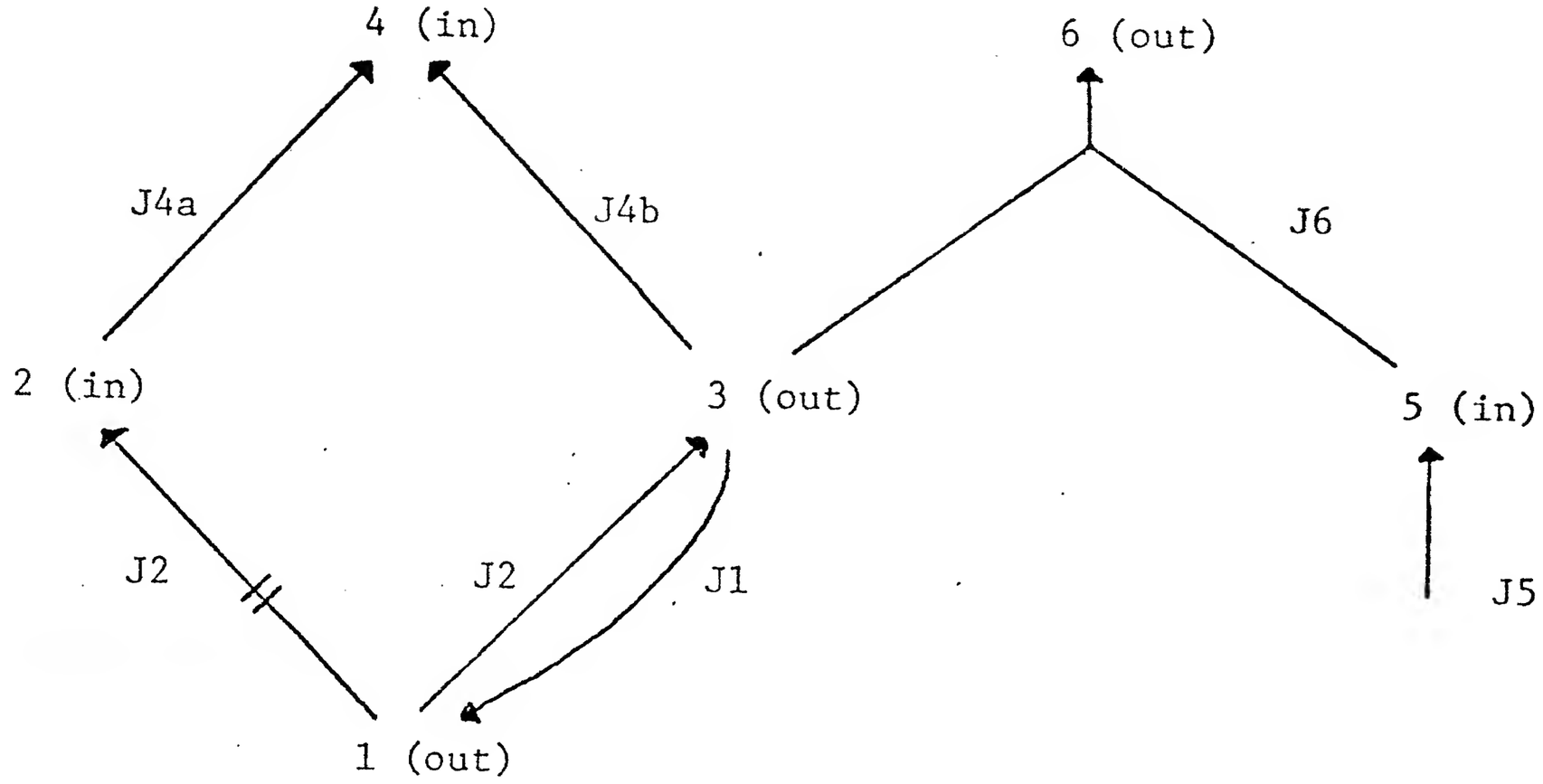

Figure 2. A depiction of the system of Figure 1. All arrows represent justifications. The uncrossed arrows represent inlists, and only the crossed line of J2 represents an outlist. We always visualize support relationships as pointing upwards.

Dependency	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6
SUPPORT-STATUS	out	in	out	in	in	out
SUPPORTING-JUSTIFICATION	–	J2	–	J4a	J5	–
SUPPORTING-NODES	3	1	1	2	–	3
ANTECEDENTS	–	1	–	2	–	–
FOUNDATIONS	–	1	–	1,2	–	–
ANCESTORS	1,3	1,3	1,3	1,2,3	–	1,3
CONSEQUENCES	2,3	4	1,4,6	–	6	–
AFFECTED-CONSEQUENCES	2,3	4	1,6	–	–	–
BELIEVED-CONSEQUENCES	2	4	–	–	–	–
REPERCUSSIONS	1,2,3,4,6	4	1,2,3,4,6	–	–	–
BELIEVED-REPERCUSSIONS	2,4	4	–	–	–	–

Figure 3. A table of all the dependency relationships implicit in the system of Figure 1. Dashed entries are empty. All other entries are lists of nodes in the dependency relationship to the node given at the top of the column.